Bcfg2 Architecture

Bcfg2 is based on a client-server architecture. The client is responsible for interpreting (but not processing) the configuration served by the server. This configuration is literal, so no local process is required. After completion of the configuration process, the client uploads a set of statistics to the server. This section will describe the goals and then the architecture motivated by it.

Goals

- **Model configurations using declarative semantics.**
  Declarative semantics maximize the utility of configuration management tools; they provide the most flexibility for the tool to determine the right course of action in any given situation. This means that users can focus on the task of describing the desired configuration, while leaving the task of transitioning clients states to the tool.
- **Configuration descriptions should be comprehensive.**
  This means that configurations served to the client should be sufficient to reproduce all desired functionality. This assumption allows the use of heuristics to detect extra configuration, aiding in reliable, comprehensive configuration definitions.
- **Provide a flexible approach to user interactions.**
  Most configuration management systems take a rigid approach to user interactions; that is, either the client system is always correct, or the central system is. This means that users are forced into an overly proscribed model where the system asserts where correct data is. Configuration data modification is frequently undertaken on both the configuration server and clients. Hence, the existence of a single canonical data location can easily pose a problem during normal tool use. Bcfg2 takes a different approach.

The default assumption is that data on the server is correct, however, the client has option to run in another mode where local changes are catalogued for server-side integration. If the Bcfg2 client is run in dry run mode, it can help to reconcile differences between current client state and the configuration described on the server. The Bcfg2 client also searches for extra configuration; that is, configuration that is not specified by the configuration description. When extra configuration is found, either configuration has been removed from the configuration description on the server, or manual configuration has occurred on the client. Options related to two-way verification and removal are useful for configuration reconciliation when interactive access is used.

- Plugins and administrative applications.
- Incremental operations.

The Bcfg2 Client

The Bcfg2 client performs all client configuration or reconfiguration operations. It renders a declarative configuration specification, provided by the Bcfg2 server, into a set of configuration operations which will, if executed, attempt to change the client's state into that described by the configuration specification. Conceptually, the Bcfg2 client serves to isolate the Bcfg2 server and specification from the imperative operations required to implement configuration changes.

This isolation allows declarative specifications to be manipulated symbolically on the server, without needing to understand the properties of the underlying system tools. In this way, the Bcfg2 client acts as a sort of expert system that "knows" how to implement declarative configuration changes.

The operation of the Bcfg2 client is intended to be as simple as possible. The normal configuration process consists of four main steps:

- **Probe Execution**
  During the probe execution stage, the client connects to the server and downloads a series of probes to execute. These probes reveal local facts to the Bcfg2 server. For example, a probe could discover the type of video card in a system. The Bcfg2 client returns this data to the server, where it can influence the client configuration generation process.
- **Configuration Download and Inventory**
  The Bcfg2 client now downloads a configuration specification from the Bcfg2 server. The configuration describes the complete target state of the machine. That is, all aspects of client configuration should be represented in this specification. For example, all software packages and services should be represented in the configuration specification. The client now performs a local system inventory. This process consists of verifying each entry present in the configuration specification. After this check is completed, heuristic checks for configuration not included in the configuration specification. We refer to this inventory process as 2-way validation, as first we verify that the client contains all configuration that is included in the specification, then we check if the client has any extra configuration that isn't present. This provides a fairly rigorous notion of client configuration congruence. Once the 2-way verification process has been performed, the client has built a list of all configuration entries that are out of spec. This list has two parts: specified configuration that is incorrect (or missing) and unspecified configuration that should be removed.
- **Configuration Update**
  The client now attempts to update its configuration to match the specification. Depending on options, changes may not (or only partially) be performed. First, if extra configuration correction is enabled, extra configuration can be removed. Then the remaining changes are processed. The Bcfg2 client loops while progress is made in the correction of these incorrect configuration entries. This loop results in the client being able to accomplish all it will be able to during one execution. Once all entries are fixed, or no progress is being made, the loop terminates.Once all configuration changes that can be performed have been, bundle dependencies are handled. Bundle groupings result in two different behaviors. Contained entries are assumed to be inter-dependant. To address this, the client re-verifies each entry in any bundle containing an updates configuration entry. Also, services contained in modified bundles are restarted.
- **Statistics Upload**
  Once the reconfiguration process has concluded, the client reports information back to the server about the actions it performed during the reconfiguration process. Statistics function as a detailed return code from the client. The server stores statistics information. Information included in this statistics update includes (but is not limited to):
    - Overall client status (clean/dirty)
    - List of modified configuration entries
    - List of uncorrectable configuration entries

Architecture Abstraction

The Bcfg2 client internally supports the administrative tools available on different architectures. For example, rpm and apt-get are both supported, allowing operation of Debian, Redhat, SUSE, and Mandriva systems. The client toolset is specified in the configuration specification. The client merely includes a series of libraries which describe how to interact with the system tools on a particular platform.

Three of the libraries exist. There is a base set of functions, which contain definitions describing how to perform POSIX operations. Support for configuration files, directories, and symlinks are included here. Two other libraries subclass this one, providing support for Debian and rpm-based systems.

The Debian toolset includes support for apt-get and update-rc.d. These tools provide the ability to install and remove packages, and to install and remove services.

The Redhat toolset includes support for rpm and chkconfig. Any other platform that uses these tools can also use this toolset. Hence, all of the other familiar rpm-based distributions can use this toolset without issue.

Other platforms can easily use the POSIX toolset, ignoring support for packages or services. Alternatively, adding support for new toolsets isn't difficult. Each toolset consists of about 125 lines of python code.

The Bcfg2 Server

The Bcfg2 server is responsible for taking a network description and turning it into a series of configuration specifications for particular clients. It also manages probed data and tracks statistics for clients.

The Bcfg2 server takes information from two sources when generating client configuration specifications. The first is a pool of metadata that describes clients as members of an aspect-based classing system. That is, clients are defined in terms of aspects of their behavior. The other is a file system repository that contains mappings from metadata to literal configuration. These are combined to form the literal configuration specifications for clients.

The Configuration Specification Construction Process

As we described in the previous section, the client connects to the server to request a configuration specification. The server uses the client's metadata and the file system repository to build a specification that is tailored for the client. This process consists of the following steps:

- **Metadata Lookup**
  The server uses the client's IP address to initiate the metadata lookup. This initial metadata consists of a (profile, image) tuple. If the client already has metadata registered, then it is used. If not, then default values are used and stored for future use. This metadata tuple is expanded using some profile and class definitions also included in the metadata. The end result of this process is metadata consisting of hostname, profile, image, a list of classes, a list of attributes and a list of bundles.
- **Abstract Configuration Construction**
  Once the server has the client metadata, it is used to create an abstract configuration. An abstract configuration contains all of the configuration elements that will exist in the final specification without any specifics. All entries will be typed (ie the tagname will be one of Package, ConfigurationFile, Service, !Symlink, or Directory) and will include a name. These configuration entries are grouped into bundles, which document installation time interdependencies.

- **Configuration Binding**
  The abstract configuration determines the structure of the client configuration, however, it doesn't contain literal configuration information. After the abstract configuration is created, each configuration entry must be bound to a client-specific value. The Bcfg2 server uses plugins to provide these client-specific bindings. The Bcfg2 server core contains a dispatch table that describes which plugins can handle requests of a particular type. The responsible plugin is located for each entry. It is called, passing in the configuration entry and the client's metadata. The behavior of plugins is explicitly undefined, so as to allow maximum flexibility. The behaviours of the stock plugins are documented elsewhere in this manual. Once this binding process is completed, the server has a literal, client-specific configuration specification. This specification is complete and comprehensive; the client doesn't need to process it at all in order to use it. It also represents the totality of the configuration specified for the client.

The Literal Configuration Specification

Literal configuration specifications are served to clients by the Bcfg2 server. This is a differentiating factor for Bcfg2; all other major configuration management systems use a non-literal configuration specification. That is, the clients receive a symbolic configuration that they process to implement target states. We took the literal approach for a few reasons:

- A small list of configuration element types can be defined, each of which can have a set of defined semantics. This allows the server to have a well-formed model of client-side operations. Without a static lexicon with defined semantics, this isn't possible. This allows the server, for example, to record the update of a package as a coherent event.
- Literal configurations do not require client-side processing. Removing client-side processing reduces the critical footprint of the tool. That is, the Bcfg2 client (and the tools it calls) need to be functional, but the rest of the system can be in any state. Yet, the client will receive a correct configuration.
- Having static, defined element semantics also requires that all operations be defined and implemented in advance. The implementation can maximize reliability and robustness. In more ad-hoc setups, these operations aren't necessarily safely implemented.

The Structure of Specifications

Configuration specifications contain some number of clauses. Two types of clauses exist. Bundles are groups of inter-dependent configuration entities. The purpose of bundles is to encode installation-time dependencies such that all new configuration is properly activated during reconfiguration operations. T hat is, if a daemon configuration file is changed, its daemon should be restarted. Another example of bundle usage is the reconfiguration of a software package. If a package contains a default configuration file, but it gets overwritten by an environment-specific one, then that updated configuration file should survive package upgrade. The purpose of bundles is to describe services, or reconfigured software packages. Independent clauses contains groups of configuration entities that aren't related in any way. This provides a convenient mechanism that can be used for bulk installations of software.

Each of these clauses contains some number of configuration entities. Five types of configuration entities exist: ConfigurationFile, Package, SymLink, Directory, and Service. Each of these correspond to the obvious system item. Configuration specifications can get quite large; many systems have specifications that top one megabyte in size. An example of one is included in an appendix. These configurations can be written by hand, or generated by the server. The easiest way to start using Bcfg2 is to write small static configurations for clients. Once configurations get larger, this process gets unwieldy; at this point, using the server makes more sense.

Design Considerations

This section will discuss several aspects of the design of bcfg2, and the particular use cases that motivated them. Initially, this will consist of a discussion of the system metadata, and the intended usage model for package indices as well.

System Metadata

Bcfg2 system metadata describes the underlying patterns in system configurations. It describes commonalities and differences between these specifications in a rigorous way. The groups used by bcfg2's metadata are responsible for differentiating clients from one another, and building collections of allocatable configuration.

The Bcfg2 metadata system has been designed with several high-level goals in mind. Flexibility and precision are paramount concerns; no configuration should be undescribable using the constructs present in the bcfg2 repository. We have found (generally the hard way) that any assumptions about the inherent simplicity of configuration patterns tend to be wrong, so obscenely complex configurations must be representable, even if these requirements seem illogical during the implementation.

In particular, we wanted to streamline several operations that commonly occurred in our environment.

- Copying one node's profile to another node.

In many environments, many nodes are instances of a common configuration specification. They all have similar roles and software. In our environment, desktop machines were the best example of this. Other than strictly per-host configuration like SSH keys, all desktop machines use a common configuration specification. This trivializes the process of creating a new desktop machine.

- Creating a specialized version of a currently existing profile.

In environments with highly varied configurations, departmental infrastructure being a good example, "another machine like X but with extra software" is a common requirement. For this reason, it must be trivially possible to inherit most of a configuration specification from some more generic source, while being able to describe overriding aspects in a convenient fashion.

- 3. Compose several pre-existing configuration aspects to create a new profile.

The ability to compose configuration aspects allows the easy creation of new profiles based on a series of predefined set of configuration specification fragments. The end result is more agility in environments where change is the norm.

In order for a classing system to be comprehensive, it must be usable in complex ways. The Bcfg2 metadata system has constructs that map cleanly to first-order logic. This implies that any complex configuration pattern can be represented (at all) by the metadata, as first-order logic is provably comprehensive. (There is a discussion later in the document describing the metadata system in detail, and showing how it corresponds to first-order logic)

These use cases motivate several of the design decisions that we made:

- There must be a many to one correspondence between clients and groups. Membership in a given profile group must imbue a client with all of its configuration properties.

Package Management

The interface provided in the bcfg2 repository for package specification was designed with automation in mind. The goal was to support an append only interface to the repository, so that users do not need to continuously re-write already existing bits of specification.